The AGENT0 Manual

Mark C. Torrance
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
torrance@ai.mit.edu

October 25, 1991

This document describes an implementation of **AOP**, an interpreter for programs written in a language called **AGENT0**. **AGENT0** is a first stab at a programming language for the paradigm of Agent-Oriented Programming. It is currently under development at Stanford under the direction of Yoav Shoham. This implementation is the work of the author in collaboration with Paul A. Viola, also of MIT (viola@ai.mit.edu).

1 Introduction

AOP, *Agent Oriented Programming*, is a programming paradigm proposed by Professor Yoav Shoham of Stanford University. It imposes certain constraints on the nature of agents and of their communication. **AGENT0** is a more restrictive language, in which programs in the spirit of **AOP** can be written. Both **AOP** and **AGENT0** are described in [Shoham, 1990].

This document describes an implementation of an interpreter for agent programs written in the **AGENT0** language. This implementation purports to be a complete implementation of the **AGENT0** language as defined in Shoham's paper. This implementation is still under development, but the most recent released version should be fairly complete and accurately described by this document.

2 Obtaining AGENT0

This implementation of **AGENT0** is written in Common Lisp, and should run under any Common Lisp interpreter. It has been tested under AKCL and Allegro Common Lisp, on Sparcstations, Dec Workstations, and Macintosh computers. The code for the interpreter and the simple Command Line Interface should be portable to any Common Lisp implementation. There are additional programs which provide a nice Graphical User Interface under Xwindows. This GUI is described in the section **Xwindows GUI**.

The notation <a0> will refer to the directory in which your **AGENT0** files are stored. You should set up such a directory in a convenient place, and put all of the **AGENT0** files there.

If you are a member of the Nobotics group at Stanford, the **AGENT0** files are available in the directory ~aop/lisp/a0. You don't need to copy these; just run them from this directory. Hereafter, you should use ~aop/lisp/a0 wherever you see <a0>.

AGENT0 is also available on the Andrew File System, a national network-transparent filesystem, in the directory /afs/ir.stanford.edu/users/t/torrance/aop. Copy all files from this directory into a directory (hereafter <a0>) on your own machine, and change the variable *aop-load-path* in <a0>/load.lisp to point to the place where you put the <a0> directory. This pathname should either be absolute for your machine, or relative to the *default-pathname-defaults* of Lisp as you start it, which is the pathname of the directory you are in when you started Lisp.

Finally, **AGENT0** is available for anonymous ftp from trix.ai.mit.edu in the directory pub/aop. Make the same changes described in the preceding paragraph to get it to run on your installation.

3 Running AGENT0

To run **AGENT0**, first start up Common Lisp. The command to do this will depend on your system, but could be cl, akcl, acl, or clicking on an icon for Allegro Common Lisp on a Macintosh.

Next, load the file <a0>/load by typing (load ``load'') to Lisp. If you were not in the <a0> directory when you started Lisp, you should type the pathname of that directory to the load command, as in (load `` aop/lisp/a0/load''), which works on Nobotics lab machines.

4 The (aop) function

After you have loaded **AOP**, calling the (aop) function will start the interpreter's read-eval-print loop. The prompt will be <AGENT>, representing the fact that you are "in the context of" a predefined agent named agent. This agent has no beliefs or commitment rules built in. It is useful mainly as a place from which to interact with other agents running under the interpreter.

5 Defining an Agent

Each agent must have a different name. The program for an agent named *agent-name* is stored in the file agent-name.lisp, and consists of a call to the **defagent** macro to define the agent followed by ordinary lisp functions which implement private actions. This file should be stored in the directory referenced as *aop-agent-path* in the file load.lisp. If you wish to load agents from elsewhere, rebind this variable to point to the directory where you store your agents.

The **defagent** macro defines a new agent.

defagent *name &key :timegrain :beliefs :commit-rules* *Macro*
Defines an agent named *name*. See **BNF** at the end of this manual for the structure of the arguments to defagent. The timegrain is not currently used by the interpreter, so just put a value here as a placeholder. *beliefs* is a list of initial beliefs of the agent, and *commitrules* is a set of commitment rules which provides the main program for the agent's behavior.

A sample agent named joetriv has been provided in the `agents` directory. You can examine the file `joetriv.lisp` to get an idea of the format of this information. The next section describes how to load an agent such as joetriv into the environment.

6 Loading an Agent

To load an agent into the current environment, say an agent named joetriv, type `load joetriv` to the `<AGENT>` prompt. The prompt will change to reflect the fact that you are now within the context of the agent `joetriv`. This sets the global variable `*current-agent*` to the internal data structure associated with joetriv, so that many of the commands understood by the **AGENT0** main loop will function with respect to joetriv. For example, you could type `state (now (alive john))` to assert a fact into joetriv's beliefs. Or you could type `inform agent ((+ now (* 10 m)) (foo a b))` to send an inform message from joetriv to agent which says that the proposition `(foo a b)` becomes true ten minutes from the time the message is sent.

7 Beliefs

An agent's beliefs consist of a set of facts. Each fact is associated with a predicate and a fact-status list. This list describes the truth-values of the fact over time. An example of a fact-status list would be the following:

```
[.. U] [10:00:00 T] [Sun Nov 24 12:00:00 F]
```

This indicates that the agent believes the predicate of the fact in question became true at 10am today, and will become false at 12 noon on Nov 24 of this year. If you ask this agent whether she believes the fact at some time between these two, she will answer `t`; if you ask about some time after this range, she will answer `nil`; if you ask about a time before this range, she will answer `nil` both to queries about the fact and to queries about its negation. This is because the truth value up until 10am today is "unknown".

As described in Yoav's paper, **AGENT0** agents believe any new fact they are told. Facts, here, are really statements about the status of a proposition at a particular time. These are parsed as statements of the form `(TIME (PRED args))`, internally called `fact-patterns`. Each typically will give rise to a new fact-status record on the fact with the same proposition as was passed in the message, time equal to `TIME` (which must be bound), and truth-value taken from the presence or

`q` or `quit` or `exit`	leave the **AOP** function
`run`	begin asynchronous mode (run ticks continuously)
`walk` or `stop`	return to synchronous mode
`<return>`	on a line by itself, runs one tick in synchronous mode
`now`	print the current time in 24-hour format
`load <agent>`	loads files `<agent>.aop` and `<agent>.lsp`
`go <agent>`	make `<agent>` the `*current-agent*`
`inform <agent> (TIME PROP)`	`*current-agent*` informs `<agent>` of `<fact>`
`request <agent> <act>`	`*current-agent*` requests `<agent>` to perform `<act>`
`beliefs`	list all beliefs of `*current-agent*`
`cmtrules`	list all commitment rules of `*current-agent*`
`incoming`	list all incoming messages of `*current-agent*` (to be processed at beginning of next tick)
`bel? (TIME PROP)`	tells whether `*current-agent*` believes `PROP` is true at `TIME` (`now` can be used as a `TIME` to indicate the current time. Functions of `now` can also be used, such as `(+ now (* 2 m))` for 2 minutes later than the current time)
`state (TIME PROP)`	assert this fact as a belief of `*current-agent*`
`clrbels`	remove all beliefs of `*current-agent*`
`cmtrule [ , , , ]`	add a new commit-rule
`showmsgs`	turn on display of messages as they are sent
`noshowmsgs`	turn off display of messages as they are sent
`<any-lisp-form>`	let Lisp evaluate `<form>`

Figure 1: Commands for the command-line interface

absence of a `not` before the `PRED`. Some special statements are allowed in place of `TIME` here; see Section 11 below for details.

8 Commitments

A commitment is just a particular kind of proposition which is stored in an agent's beliefs database. An agent can come to have a commitment either as a result of firing a commitment rule, triggered by some incoming request message, by taking the action of committing to do some other action, or by simply asserting the commitment into his beliefs. A commitment can be unrequested by the agent to whom it is made. An agent can commit to herself; this is considered a "choice".

Each tick, an agent performs all of her commitments which have matured. A commitment it is unasserted from the beliefs database (i.e., asserted with truth-value **FALSE**), at the time the commitment is performed. This gives the agent a record of the time at which she actually carried out the commitment.

In the most recent version of the interpreter, commitments are performed when the current time is equal to or past the time expressed in the action. This means that when a commitment rule fires and installs a commitment, that commitment will get acted upon later that same tick if the action refers to the current time or a time in the past. In order to make it possible to reason about things to which an agent has been committed in the past, the commitment is actually asserted with truth value **FALSE** one second later than the current time, so that commitments to immediate action will remain true with some duration in the belief history. This is correct only when the time between ticks is more than one second.

9 Capabilities

The **AGENT0** specification calls for a database of capabilities to be checked against automatically each time an agent considers making a commitment. This current implementation of **AGENT0** does not include any capability database or checking of such a database.

10 Messages

Agents can send each other `REQUEST` and `INFORM` messages. The syntax is as follows:

```
<AGENT> inform joetriv (now (i_am_cool))

<AGENT> request joetriv (do (+ now (* 5 m)) (becool))
```

The "now" in the message refers to the moment when the message was sent, not the moment when it was received.

Agents can also send messages from within private action functions. This facility may be used to, among other things, send multiple messages as a result of a single commitment rule firing. Functions to send messages from within private actions are described below in the section **Private Actions**.

The next version of this interpreter will include support for a standardized message-passing format in terms of files or UNIX sockets, so that agents running under this implementation can communicate with agents running under other implementations or on other machines.

11 Time

AGENT0 can be run in either synchronous or asynchronous mode. The guarantee made in general of **AGENT0** programs is that they keep their commitments by the time they mature. In this implementation, the agents always perform their commitments during the first tick which is begun after those commitments mature. If the simulation is being run in discrete, user-prompted tick cycles, as it will be when the user wants to interactively send messages and inspect agents from the command-line interface, then it is up to the user to hit return often enough that the agents meet their commitments in a timely manner. To run the simulator in an asynchronous mode, type `run` to the prompt. An asterisk will appear before the prompt to indicate that ticks are being run. To return to the synchronous mode, type `q` or `quit` to the prompt.

I have chosen to print real times to varying degrees of specificity, depending on how far the time is from the current time. Thus, if the time is in the format `HH:MM:SS`, it is some time during the current day, printed in 24-hour time format. If the time is within the next week, and in the same month as the current day, the day of the week is given with the time for display purposes. If it is not, but it is within the same year, then all but the year are shown. Otherwise, a full display of `DAY MONTH DATE HH:MM:SS YEAR` is shown.

When using the Lisp syntax, users specify times by using Lisp functions. These functions will operate on time in the internal (integer) format. So `(+ ?time 30)` is a time 30 seconds later than the time to which `?time` is bound. Several useful constants are defined to make it easier to specify relative times. These include `m` for one minute, `h` for one hour, `day`, `week`, and `yr`. I do not yet provide functions for specifying absolute times, but I plan to soon. For now, you can generate a universal time integer by using the Lisp function `(encode-universal-time args)`. Its syntax is as follows:

```
------------------------------------------------------------------------
ENCODE-UNIVERSAL-TIME [Function] Args: (second minute hour date month
year &optional (timezone -9))
------------------------------------------------------------------------
```

The correct time zone to use on the West Coast of the United States is 7.

Users and agent programs can also use the word `now` to refer to the current time. Thus, statements such as `state ((+ now (* 2 m)) (i-am-cool))` are acceptable commands to the `<AGENT>` prompt.

12 Private Actions

Private actions are merely Lisp function calls. Various functions are provided to make it easy to write useful private actions. These functions are detailed here.

lisp-parse-query *fact* *Function*
Parses a fact, which may contain references to 'now' or free variables, and returns the internal data structure called a pattern. You should not need this, since the functions below have been modified to call it themselves. At the time of execution, 'now' will be replaced by the current time.

get-belief *agent-name pattern* *Function*
A *pattern* here is just a proposition which may have variables for some of its terms. This function determines whether the pattern unifies with any of the beliefs of the agent. If it does, it returns the variable bindings resulting from the first such unification (possibly nil). If it doesn't, it returns fail. An example of the use of this function is `(get-belief 'joetriv '(on ?x ?y))`, which could return `((x . a) (y . b))` indicating that a belief of the form `(on a b)` was matched in joetriv's beliefs database.

get-all-beliefs *agent-name pattern* *Function*
Returns a list of all binding lists resulting from successful unification of the pattern with the beliefs of the agent. If no matches were made, it will return nil, the empty list. An example of the use of this function is `(get-all-beliefs 'joetriv (lisp-parse-query '(on ?x ?y)))`, which might return `(((x . a) (y . b)) ((x . c) (y . d)))` indicating that beliefs of the form `(on a b)` and `(on c d)` were both matched in joetriv's beliefs database.

delete-belief-pattern *agent-name pattern* *Function*
Completely deletes all beliefs matching the pattern from the agent's beliefs database. Note that this may drastically modify the belief structure; use it with discretion.

send-inform *from-agent-name to-agent-name fact* *Function*
Sends an inform message to the agent named *to-agent-name* concerning *fact*. Not that you do *not*

need to parse the *fact* first with **lisp-parse-fact**. An example of the use of this function is

```
(send-inform-message 'me 'you '(now (on a b)))
```

which sends a message from me to you regarding the truth of the proposition `(on a b)` at the time the command is executed.

send-request *from-agent-name to-agent-name act* *Function*
Sends a request message to the agent named *to-agent-name* to perform *act*. Note that you do *not* need to parse the *act* first with **lisp-parse-act**. An example of the use of this function is

```
(send-request-message 'me 'you '(do now (your-private-action)))
```

which sends a message from me to you requesting you to perform your private action.

bel? *agent-name fact* *Function*
This function returns T or NIL depending on whether the agent believes the fact. The fact should be of the form `(time (proposition))`. Note that for facts not explicitly in the database at all agents believe neither the fact nor its negation. Note that this function does *not* require you to parse the fact first with **lisp-parse-fact**.

inform *agent-name predicate time* *Function*
This should really be called "insert-belief". This function modifies the beliefs of the named agent without sending an inform message. This modification takes place immediately, and may cause unexpected behavior depending on when within the tick it is executed, and whether other rules or commitments depend on the belief that was changed.

inform-fact *agent-name fact* *Function*
This function also modifies the beliefs of the named agent directly. Its only difference is that it takes a fact, rather than separate predicate and time. Note that this function does *not* require you to parse the fact first with **lisp-parse-fact**.

now *Function*
Returns the current time as an integer.

time-string *time* *Function*
Returns a string representing the *time* in a display format, relative to the current time. May be useful for debugging.

13 Xwindows GUI

The latest addition to is a graphical user interface which runs under Xwindows. To use it, you must be using Lisp with CLX loaded. If CLX is not already loaded into your Lisp image, load it before loading AOP. To run the graphics, simply type `g` or `graphics` to the `<AGENT>` prompt. The buttons are fairly self-explanatory.

14 Examples

For practice, try running through a few examples in **AGENT0** to get the hang of using the interpreter and watching messages. This is an example of an interaction that exercises a subset of the part of **AGENT0** which currently works.

```
<AGENT> load joetriv
Defining agent "JOETRIV"
Parsing file aop/joetriv.aop now

LOADED

<AGENT> inform joetriv (100 (on a b))  ; This means at time 100, (on a b)
JOETRIV will be informed next tick.

<JOETRIV> beliefs

<JOETRIV>               ; (press return to run a tick)

<JOETRIV> beliefs

(ON A B)   [.. U] [100 T]
<JOETRIV> state (200 (not (on a b)))
Belief added.

<JOETRIV> beliefs
(ON A B)   [.. U] [100 T] [200 F]

<JOETRIV> bel? (150 (on a b))
T

<JOETRIV> bel? (-500 (on a b))
NIL
```

```
<JOETRIV> bel? (-500 (not (on a b)))
NIL

<JOETRIV> agent

<AGENT> inform joetriv (now (i_am_cool))
JOETRIV will be informed next tick.

<AGENT>

<AGENT> request joetriv (do (+ now m) (i_am_cool))   ; one minute from now
JOETRIV will be requested next tick.

<AGENT>

<AGENT> run

*<AGENT>          ; * indicates asynchronous mode

<just under one minute passes>

This is the cool Joe Triv Agent.

q   ; user types q to quit run-mode

<AGENT> q

<cl>
```

References

[Shoham, 1990] Y. Shoham. Agent Oriented Programming. Technical Report STAN-CS-90-1335, Stanford University, 1990.

```
<beliefs>     ::= '(<fact>*) | NIL

<commitrules> ::= '(<commitrule>*) | NIL
<commitrule>  ::= (<msgcond> <mntlcond> <agent> <action>)


<msgcond>     ::= <msgconj>    | (OR <msgconj>*)
<msgconj>     ::= <msgpattern> | (AND <msgpattern>*)
<msgpattern>  ::= (<agent> INFORM  <fact>)   |
                  (<agent> REQUEST <action>) |
                  (NOT <msgpattern>) | NIL

<mntlcond>    ::= <mntlconj>    | (OR <mntlconj>*)
<mntlconj>    ::= <mntlpattern> | (AND <mntlpattern>*)
<mntlpattern> ::= (B <fact>) | (CMT <action>) | (NOT <mntlpattern>) | NIL

<action>      ::= (DO        <time> <privateaction>)  |
                  (INFORM    <time> <agent> <fact>)   |
                  (REQUEST   <time> <agent> <action>) |
                  (UNREQUEST <time> <agent> <action>) |
                  (REFRAIN   <action>)                |
                  (IF        <mntlcond> <action>)

<fact>        ::= (<time> (<predicate> <arg>*))

<time>        ::= <integer> | now | <time-constant> |
                  (+ <time> <time>) | (- <time> <time>) |
                  (* <integer> <time>)   ; Time may be a <variable> when
                                         ; it appears in a commitment rule

<time-constant> ::= m   |                ; Minute (= 60 sec/min)
                    h   |                ; Hour   (= 3600 sec/hour)
                    day |                ; day    (= sec/day)
                    yr                   ; year   (= sec/year)

<predicate>   ::= <alphanumeric_string>
<arg>         ::= <alphanumeric_string> | <variable>

<variable>    ::= ?<alphanumeric_string>
```

Figure 2: BNF for Agent0 Programs